MASSACHUSETTS INSTITUTE OF TECHONOLOGY
Project MAC
Cambridge, Massachusetts

Artificial Intelligence Project
Memo 63

Memorandum MAC-M-128
December 27,1963

SECONDARY STORAGE IN LISP*

by Daniel J. Edwards

Paper to be presented at the First International LISP Conference, Mexico City, Mexico, December 30 - January 3, 1964.

ABSTRACT

A principal limitation of LISP processors in many computations is that of inadequate primary random-access storage. This paper explores several methods of using a secondary storage medium (such as drums, disk files or magnetic tape) to augment primary storage capacity and points out some limitations of these methods.

---

*Work reported herin was supported by MIT Project MAC and sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC

Artificial Intelligence Project
Memo 63

Memorandum MAC-M-128
December 27, 1963

SECONDARY STORAGE IN LISP

by Daniel J. Edwards

A principal limitation of the LISP language, when performing large computations, is saturation of primary random-access storage with active material, both functions (programs) and list structures. When saturation occurs, the computation is forced to terminate and one wonders to what extent secondary storage media (such as drum, disk file or magnetic tape) can be used to allow the computation to continue. In general there is no reason to suppose that anything short of more primary storage can help, except at the cost of a reduction in the computation rate by several orders of magnitude--down to the random access rate of a secondary storage media. However, in some cases one can make better use of the primary storage medium while in other cases use of secondary storage may be worthwhile to allow the completion of a computation at a somewhat higher cose.

This paper will investigate several methods of using secondary storage to augment the computing capacity of a given LISP Processor. Most of these remarks will be directed toward a LISP processor on the IBM 7094, but may be generalized to LISP processors on other machines or other list processing languages. In particular the primary storage media will be called core and the secondary storage will be called tape, disk file or drum storage. It should be noted, however, that many of these ideas require a gross rewriting of the current 7094 LISP 1.5 processor if they are to be successfully implemented.

Lisp Functions

A. Library Tape

A more efficient use of core storage would be made if the LISP user could specify those LISP functions he intended to use in the current computation. These functions could then be called from secondary storage such as a library tape and the room left in core after the functions are loaded could be allocated between list structure storage and pushdown list. The LISP library tape could store compiled functions, functions as S-expressions, or both. Each function on the library tape would specify its name and the subfunctions it needed and loading could take place in a manner similar to the Fortran Monitor RSS library tape. This method still restricts the user to one core-full of active material at any one time and this core-full must contain all the functions he will use during the computation whether currently active or not.

B. Ring Buffer Function Storage

This method of storing functions is intended for use with a relatively fast drum as secondary storage. It allows the user to have as many active functions in compiled form as he wishes.

First a portion of core is set aside as a program ring buffer. Each compiled program is required to keep all of its temporary storage on the pushdown list and all function calls and returns must be made through a fixed transfer vector which indicates which functions are in the ring buffer at any one time. When a function is called which is not in the buffer, it is fetched from the drum and read into successively higher locations in the buffer. If the top of the buffer is reached the program starts over at the bottom. Since all temporary storage of active functions is on the

pushdown list, currently active functions need never be read out. In the worst case (a cyclic set of functions which will not fit simultaneously into the ring buffer), the function execution time will approach the drum rotation speed. However, in typical LISP computations much time might elapse before a function is called that is not in the ring buffer and if all the needed functions fit into the ring buffer, the computation will run at full machine speed.

The ring buffer method might be combined with a compacting garbage collector à la Minsky and the ring buffer list-structure-storage boundary may be changed according to the needs of the computation.

C. Unused Function Declaration

Even if it is not desired to implement either of the above methods for function storage, one way of making better use of core storage is to eliminate the requirement that all LISP functions which may be used in a computation be in core. An example of this requirement in 7094 LISP is the character handling and array handling functions. Many computations do not use either of these features but the space must be currently set aside as there is no provision for reclaiming this space. This space could be made available if the user could declare those basic functions which he does not intend to use and let the garbage collector turn these areas into free storage. This method would involve making a slightly more complicated garbage collector, and more study on individual cases would have to be made to determine whether the gain in space would offset the increased garbage collection time and complexity. Note that this idea is implicitly contained in both the library tape and program ring buffer methods described above.

Structures and Secondary Storage

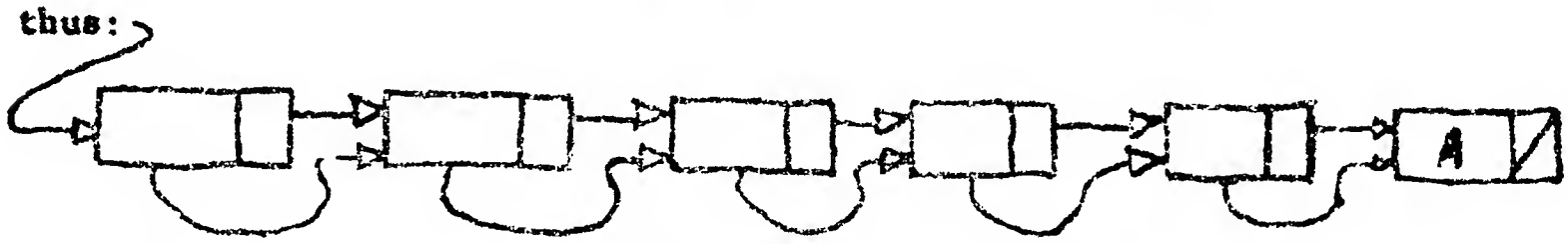
Many of the problems of using secondary storage with LISP arise from the fact that LISP freely allows lists to have sublists in common. Other list processors such as threaded lists do not allow common sublists and thus avoid some of the secondary storage problems. However, it is the author's opinion that the cure (no common sublists) is worse than the disease (hard to make use of secondary storage) because the computation rate is much slower when time and storage is spent copying and recopying sublists which could be kept in common.

A. Paging

One method of using secondary storage in LISP is to divide core into a number of blocks known as pages in manner similar to the Ferranti Atlas computer. Storage references within a given page proceed normally, while cross page references use a subroutine to determine if the referenced page is in core. If not, one of the pages in core is read out onto a drum or disk and the required page read in. This method has been programmed for list structures on the MIT PDP-1 computer with an 88,000 word drum. This computer is a 4,096 word, 18 bit, 5 microsecond machine with a LISP processor which has 800 words of free storage in core. Unfortunately, in processing list structures there is no good method known for reducing the number of cross page references and on a large problem, the computation speed is quickly reduced to the drum rotation speed. As an example, one set of functions, which the 7094 reads in in a few seconds, required 5 hours to read in with PDP-1 LISP. Other computations performed on both PDP-1 LISP and 7094 LISP show the paging method, in fact, limits the computation speed to the drum rotation speed.

B. Read-Print with Variations

Since the paging method mentioned above is wasteful of central processor time, it has been propsed to let the programmer create his own file system in secondary storage and then allow him to print out list structures with the LISP PRINT program and read them back in with READ. This method suffers because READ and PRINT are not fast programs and no method to preserve common list subexpressions is provided. The first objection may be tolerated but the second can have catastrophic results. Consider the list structure known colloquially as a BLAM list and represented in box notation thus:



A BLAM list is a valid piece of list structure in that it is not circular, but when copied without preserving common subexpressions a BLAM list of n words uses 2^n words of free storage. The BLAM list is not as far-fetched as it may seem at first since, under certain circumstances, the pair-list in the current LISP 1.5 interpreter has BLAM list characteristics.

Various methods for speeding up the READ-PRINT process such as encoding the atoms may be envisioned but failure to preserve common subexpressions in the list structure is still a big objection.

One method of preserving common subexpressions would be to allow the user to read out blocks of list structures in a compacted form such as suggested in Minsky's garbage collector paper. This method has the additional advantage of being fairly fast on read-out and very fast on reading in as free storage is kept in a block.

Any of the above methods for letting the user put out and call back list structures might allow some computations to be carried out which could not be done otherwise. An example could be the symbolic differentiation of a series which could be handled term by term with only one term in core at a time.

C. Automatic Compacting List Structure Dump

Another method to utilize secondary storage in LISP would be to program an automatic list structure dump program in conjenction with the garbage collector. Initially, the program would be allowed to run until all core is filled with active list structure. At this point the garbage collector would mark backwards along the most recent part of the pushdown list until one-half of available core storage is marked. Then the list structure referred to by the rest of the pushdown list would be compacted and filed in secondary storage. This list structure would be retrieved when the pushdown list got back to the point where the filed material was needed.

This method of automatic dumping assumes that atomic symbols are always present and none of the relevant properties are changed during the course of the computation. This method also assumes, during the course of the computation, list structures are found in disjoint pieces which may or may not be the case. That is to say that it is possible that during a computation most of the active list structures are referred to by the most recent items on the pushdown list. It should be noted that items referred to on the older part of the pushdown list cannot be dumped if they have subexpressions in common items referred to on the recent part of the

pushdown list. In order to dump the older items either the common sub-l expressions would have to be copied--which is undesirable from a storage point of view--or the common subexpressions would have to be exactly preserved which is quite hard when using a compacting garbage collector.

D. Look Ahead Methods

The last and by far the most speculative method for using secondary storage would be in conjunction with the compiler. If during the course of a computation, certain relatively fixed lists, such as dictionaries or function tables, could be declared to be secondary storage lists, the compiler could then perform a computation look ahead so that the compiled program would start getting items from secondary storage via data channels before they were actually needed. In this was the central processor could be kept running at almost full speed.

E. Conclusion

As above methods tend to show, secondary storage is rather hard to use for storing list structures during a LISP computation. It is the author's opinion that the trade-off factor between primary and secondary storage is about 30 to 1 in favor of more primary storage (i.e., 1 unit of primary storage is worth 30 units of secondary storage). However, in situations where more primary storage is not available, secondary storage may be used to some advantage in LISP.